CSC

ICT Solutions for Brilliant Minds

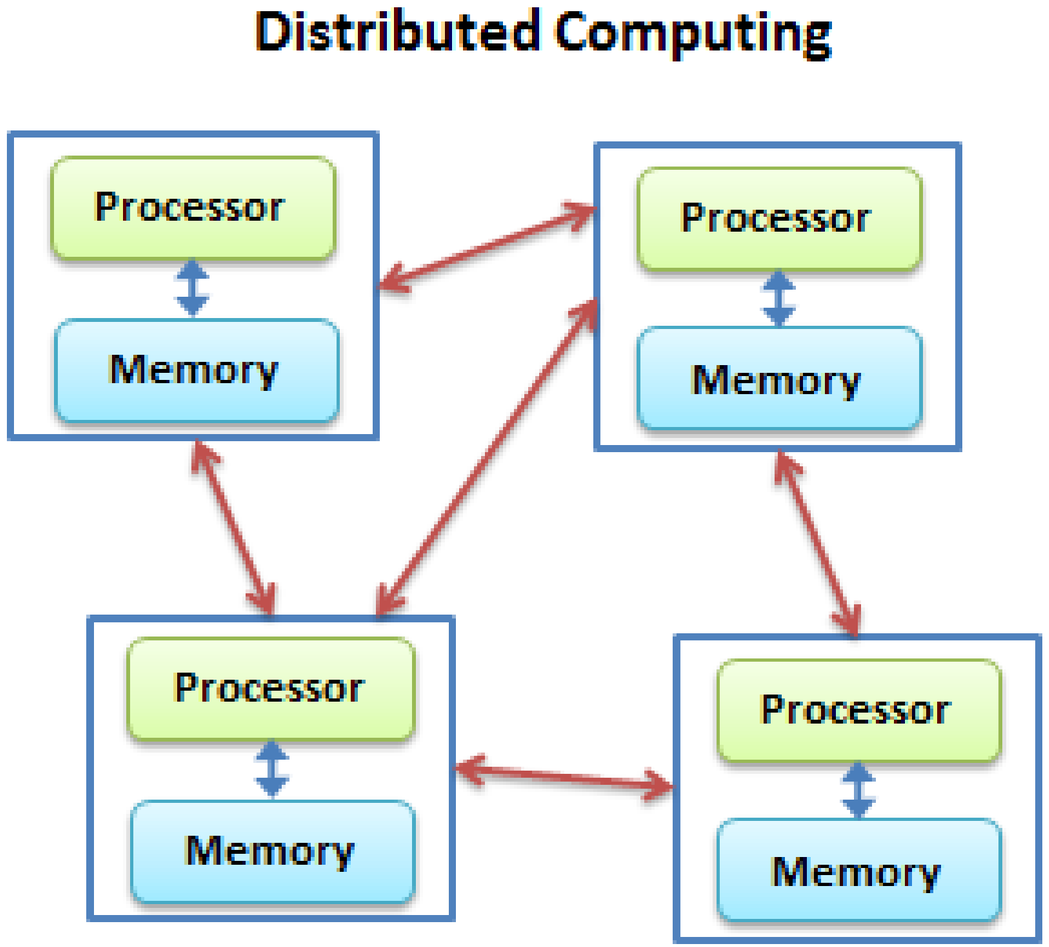
GPU Programming. When, Why and How?

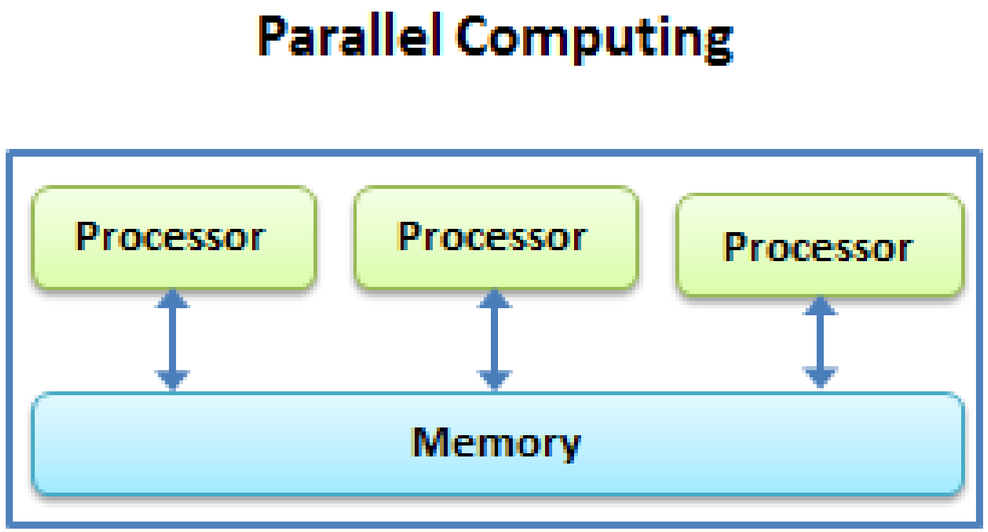
2024

ENCCS Training

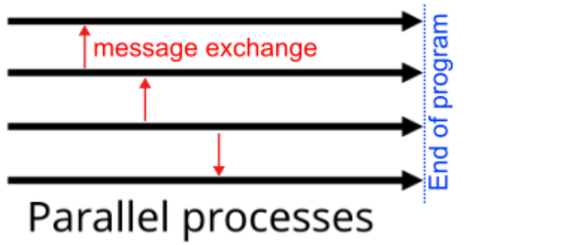
GPU programming concepts

Distributed- vs. Shared-Memory Computing

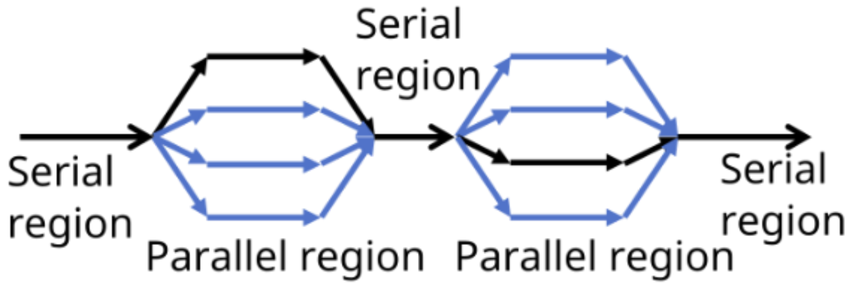




Processes and Threads



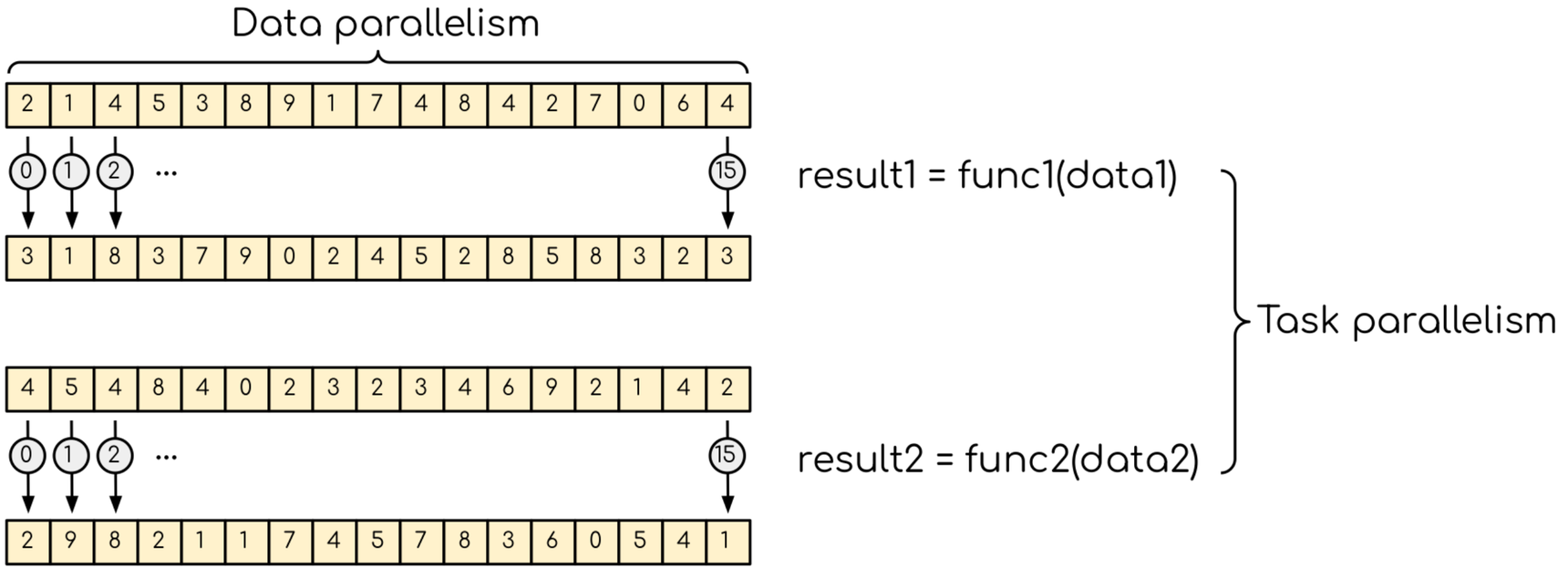
Distributed memory parallelism



Shared memory parallelism

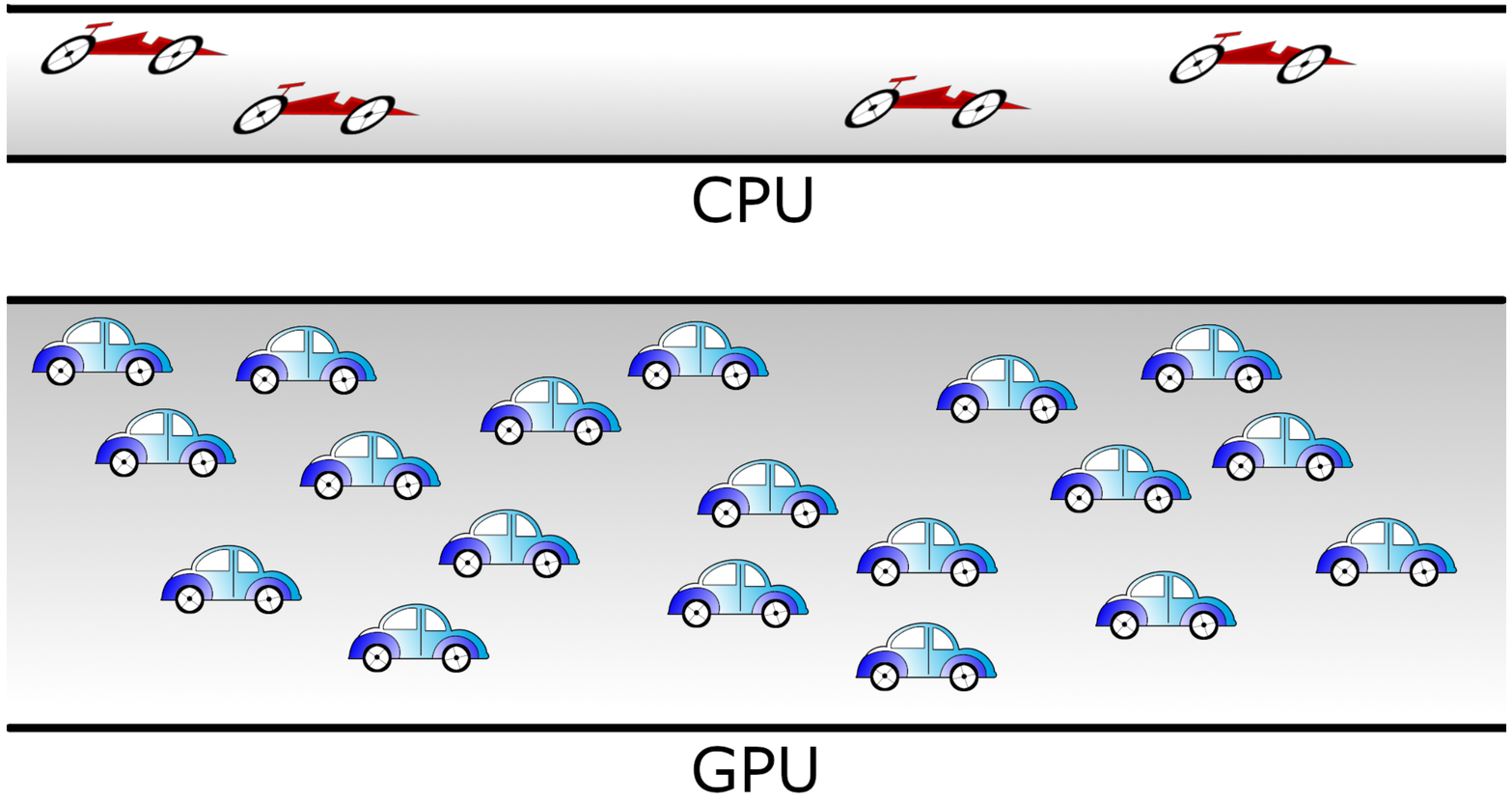
- **Process**-based parallel programming model is used on distributed memory machines.
  - independent execution units which have their *own memory* address spaces.
  - life time of the program.
- **Thread** based parallelism is used on shared memory architectures.
  - light execution units; are created/destroyed at a relatively small cost.
  - own state information, but they *share* the *same memory* address space.

Exposing Parallelism



- The **data parallelism** is when the same operation applies to multiple data (e.g. multiple elements of an array are transformed).
- The **task parallelism** implies that there are more than one independent task that, in principle, can be executed in parallel.

GPU Execution Model



Cars and roads analogy for the CPU and GPU behavior. The compact road is analogous to the CPU (low latency, low throughput) and the broader road is analogous to the GPU (high latency, high throughput).

Example: axpy

Serial cpu code of `y=y+a*x`:

- have a loop going over the each index

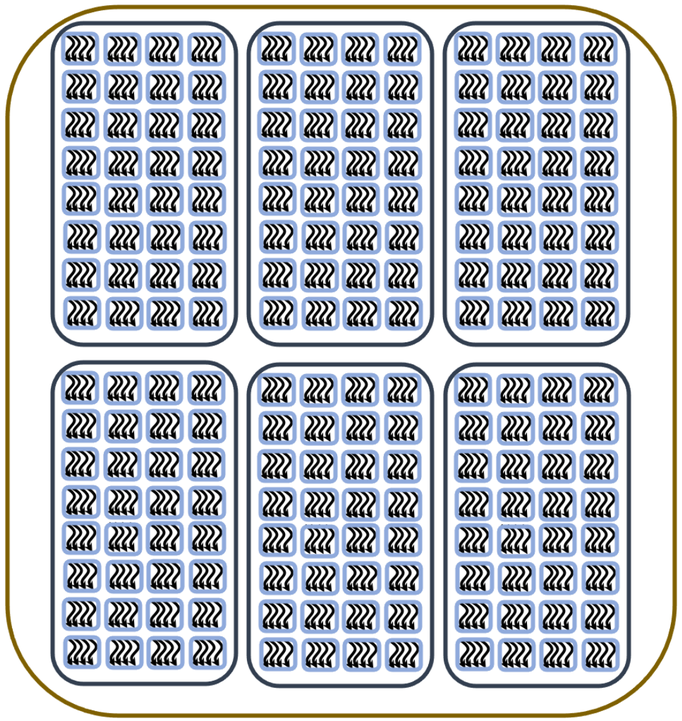
```
void axpy_(int n, double a, double *x, double *y)
{
    for(int id=0;id<n; id++) {
        y[id] += a * x[id];
    }
}
```

On an accelerator:

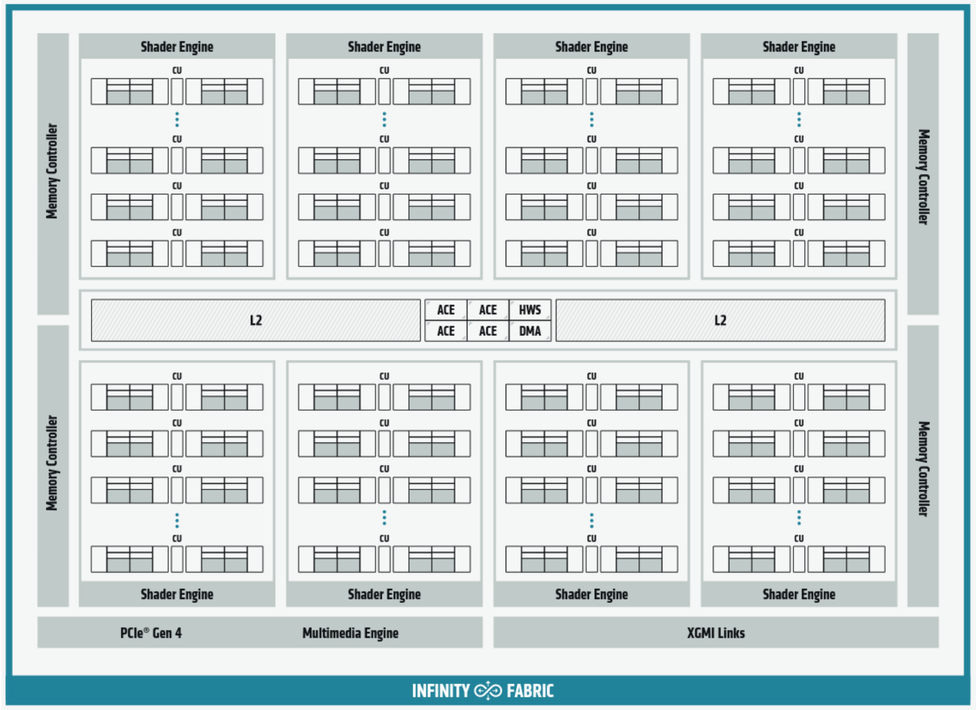
- no loop
- we create instances of the same function, **kernels**

```
GPU_K void ker_axpy_(int n, double a, double *x, double *y, int id)
{
        y[id] += a * x[id]; // id<n
}
```

Grid of CUDA Threads



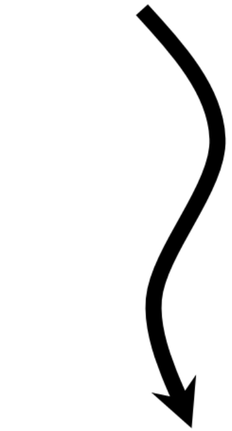
A grid of cuda threads executing the same **kernel**



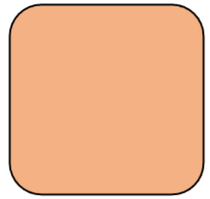
AMD Instinct MI100 architecture (source: AMD)

- Each thread executes the same kernel computing different elements of data.
- There is no global synchronization or data exchange.

CUDA Thread



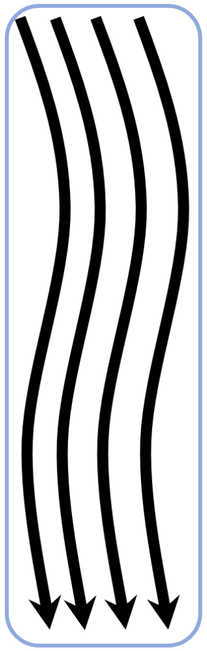
CUDA THREAD

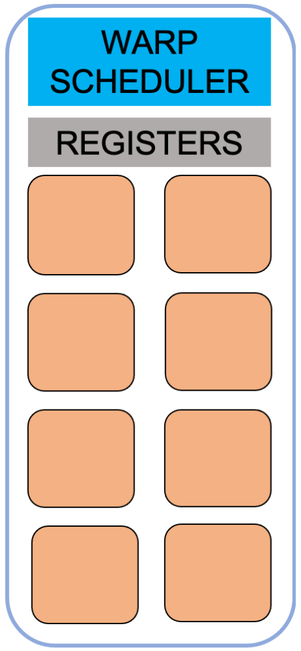


CUDA CORE

- The **CUDA threads** are very light execution contexts, containing all information needed to execute a stream of instructions.
- Each **CUDA thread** processes different elements of the data.
- Each **CUDA thread** has its own state information

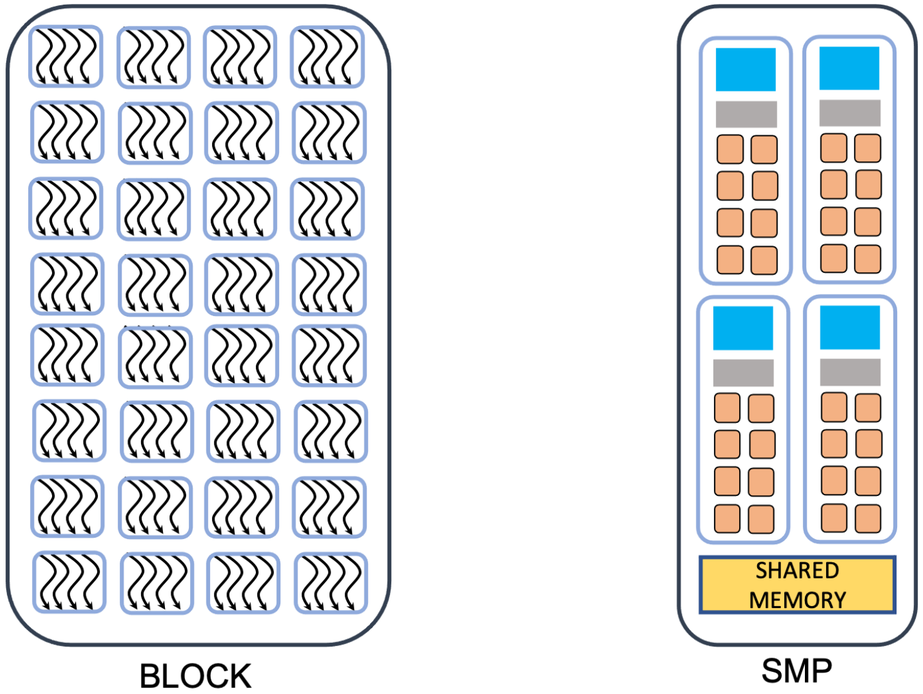
Warp





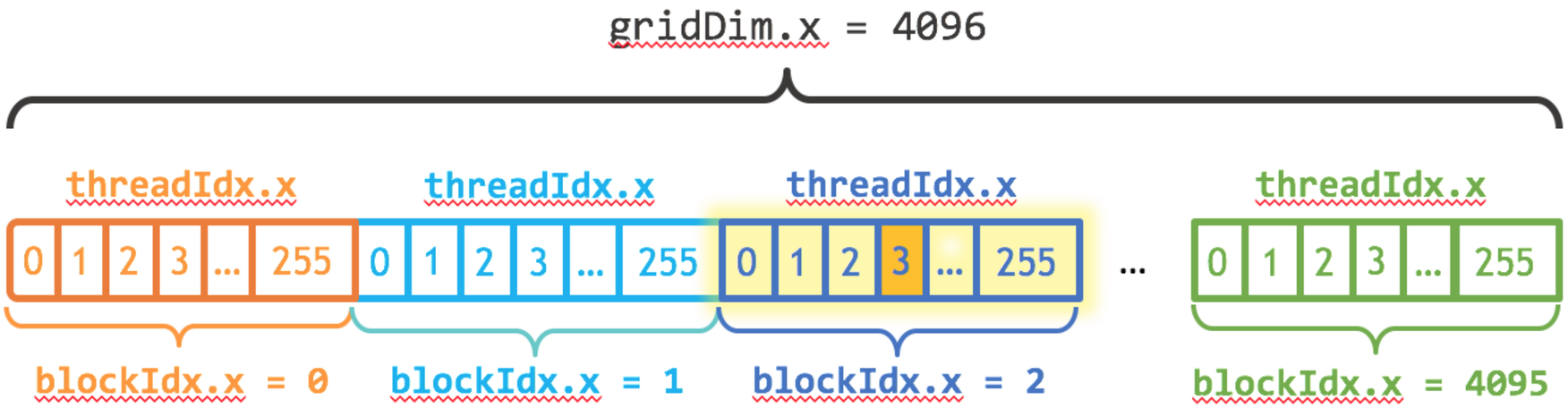
- The CUDA threads are physically locked into **warp**s, currently of size 64 for AMD and 32 for Nvidia.
- All threads in the **warp** have to execute the same instruction.
- The memory accesses are done per warp.

Block of Threads



- The threads are divided in groups of fixed size, (max 1024 for some GPUS).
- Each block is assign to a SMP and it can not be split.
- Synchronization and data exchange is possible inside a block.

Indexing




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

Terminology

NVIDIA/HIP	OpenCL/SYCL
grid of threads	NDRange
block	work-group
warp/wavefront	sub-group
thread	work-item
local memory	private memory
shared memory/local data share	local memory
threadIdx.{x,y,z}	get_local({2,1,0})
blockIdx.{x,y,z}	get_group_id({0,1,2})
blockDim.{x,y,z}	get_local_size({0,1,2})/get_local_range({2,1,0})

Summary

- Parallel computing can be classified into distributed-memory and shared-memory architectures.
- Two types of parallelism that can be explored are data parallelism and task parallelism.
- GPUs are a type of shared memory architecture suitable for data parallelism.
- GPUs have high parallelism, with threads organized into warps and blocks and.
- GPU optimization involves coalesced memory access, shared memory usage, and high thread and warp occupancy. Additionally, architecture-specific features such as warp-level operations and cooperative groups can be leveraged for more efficient processing.